**DoctorantMemory’s Handy Guide**

# **Preliminaries**

## 1a. An Overview of Programs and Memory

When executing an application on a computer, numerous memory accesses occur. There are different “types” of memory, but for our purposes consider only the RAM (Random Access Memory). From now on, when we mention memory, know that we mean RAM. Memory accesses are governed by conventions such as the heap, stack, cache, etc. For simplicity, disregard them for now.

Let’s start with a simplified overview of the types of memory accesses:

1. **Instruction Fetch**: Any computer application consists of many instructions which are stored in a specific area of the memory. When an application reads the memory to fetch the next instructions to be executed, we consider the memory access to be an instruction fetch.
2. **Read**: Any read which is not an instruction fetch. For example, printing the value of a variable to the screen requires reading it.
3. **Write**: Any modification to a value stored in the RAM. For example, allocating an array and filling it with zeros requires finding an unused section of memory and writing zeros into it. Note that there is no “Instruction Place” type of memory access, since modifying the instructions while executing them is not allowed.

The memory is a continuous 1D array of fixed size. Pointers are an address of a location in memory, like an index. When we allocate 100 bytes using malloc, we get a pointer to a location in memory, such that the 100 bytes stored from that “index” onwards are only allotted to us, until we free the pointer and they become free again. Pointers have a byte sized granularity - i.e., between pointer “4” and pointer “6” there are 2 bytes (or 8 cells that can contain 0 or 1). Pointers are usually represented using hexadecimal notation, but they are still just a “fancy index”. If you are unfamiliar with hexadecimal notation, I recommend familiarizing yourself.

The CPU (Central Processing Unit) executes the instructions. These instructions are very simple - add together two numbers, copy a single integer to a location in memory, etc. Accessing the memory is slow, much slower than executing instructions. Therefore, there is a faster, more expensive type of memory located closer to the CPU. This memory is called the Cache, and it usually has multiple levels - L1, L2, etc. The more expensive and fast it is, the smaller it is. Areas in memory that are accessed repeatedly, or are expected to be accessed soon, are copied into the Cache to save time.

## 1b. What is a Trace?

A trace is a file (or multiple files stored in a folder) containing a recording of all memory accesses made by an application during its execution. Its format targets computers and not humans, therefore it is not very insightful - mostly being a compressed list of pointers and sizes of memory accesses. Yet, we must use it to generate the statistics from which we can research the execution. A trace allows us to repeatedly analyze the program’s memory accesses, long after it has been executed.

## 1c. DynamoRIO and DrCacheSim

DynamoRIO is an instrumentation framework for the development of program analysis tools. DrCacheSim is one of many tools in the DynamoRIO framework that collects instruction and memory access traces using its DrMemTrace component. Using the traces it generates, it can generate useful statistics.

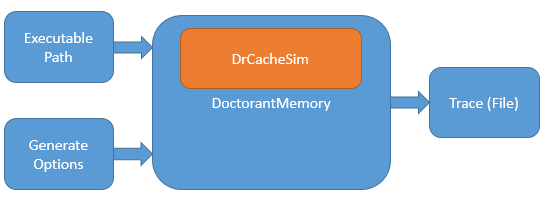
Read more about DrCacheSim here: [https://dynamorio.org/page\_DrCacheSim.html](https://dynamorio.org/page_drcachesim.html).

# **DoctorantMemory’s Structure**

By downloading DynamoRIO’s executables available from their website, we can use the DrCacheSim tool to instrument a program. We call DrCacheSim with the path to our program’s executable, along with additional parameters if needed, and DrCacheSim creates a folder containing a trace – all of the program’s memory accesses. Afterwards, we can use DrCacheSim to parse and analyze this trace, yielding statistics such as the most used memory addresses, use a cache simulator to predict the number of cache misses, etc. DoctorantMemory is a python script that receives input from the user and forwards it to DrCacheSim, applying additional processing. The main utility of DoctorantMemory is our custom trace translator: we use a DrCacheSim tool which prints a trace in a human readable format, parse the human readable trace and print the trace in our own simplified format along with useful statistics about the trace. This tool also generates a file containing all addresses in the program sorted by the number of times they have been accessed. DoctorantMemory can also call the cache simulator and address histogram tools from DrCacheSim – but no further analysis is done on the results.

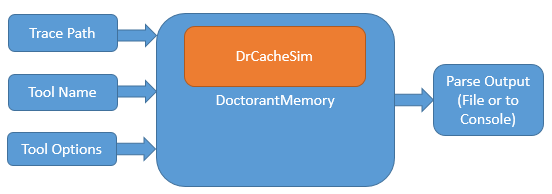
In the following figures, we demonstrate the input and output of DoctorantMemory for generating traces and parsing traces.

First, during the generation process the inputs are the path to the executable which will be ran, as well as generation options – such as the path the trace would be written to. The output is a folder containing the trace, which is one or more files. Since DrCacheSim is the “backend” of DoctorantMemory, the figures show it as being contained inside of DoctorantMemory.



During the parsing stage, the inputs are the path to an already generated trace, the name of the tool to be used, and settings such as wether instruction fetches should be ignored etc.

The output is printed to the console, but can also be saved to a file.



# **Usage**

## 3a. Generating a Trace

To analyze the execution of a program, you first need to generate a trace documenting its memory accesses.

You can generate a trace for any program of your choosing, we have included an example HelloWorld program that allocates memory, frees it, and prints hello world to the screen. To use it, it needs to be compiled first:

# **https://lh7-us.googleusercontent.com/IAeDPjYkLAdBPgqAjJw4RTqLt4b7Q74NiT4q7X8plz3bmELlFtdnj2eTt7FBZ2b1Nok_pcTzl3xPxFIixmdNbhraGG87jpkPGCTqDH2xAcCnuDb6W_eXdRZgpD_XhLq18kAScc1cFbAmJ-v-cD20ETEegI2OCX7X**

Afterwards, to generate a trace for it (or any other program) run the following command:

**python doctorant\_memory.py -operation generate -app\_path examples/HelloWorld**

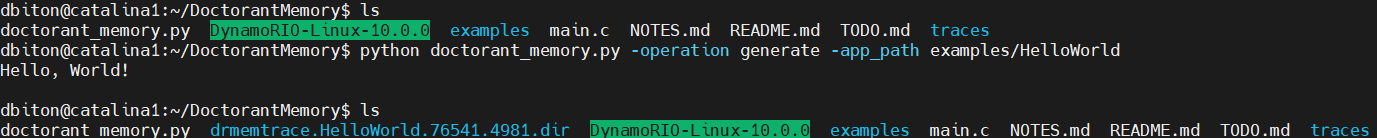
DoctorantMemory creates a file named DOCTORANT\_MEMORY\_TS, with TS being an ISO timestamp of the time the trace was created, for example: DOCTORANT\_MEMORY\_20240410T143504Z. This file contains the output of the executable, which is also printed to the console.

This file is automatically deleted, if you would like to keep it, you can use the **-keep\_logs** flag:

**python doctorant\_memory.py -operation generate -app\_path examples/HelloWorld –keep\_logs**

The trace information is written to a folder created by DrCacheSim in the current working directory.

As you can see below, after we ran the command a trace folder was created:



You can set the path where the folder would be created, with missing folders created if needed:

**python .\doctorant\_memory.py -operation generate -app\_path ./HelloWorld.exe -trace\_path traces**

The folder traces would be created if it does not exist, as well as any more needed folders.

We have included the trace output in the "example trace" folder, in case you are having trouble with generating it.

## 3b. Parsing Tools

Below we present our parsing tools, built on the base of DrCacheSim. The first 3 tools presented simply invoke DrCacheSim, while the fourth tool employs further processing. In case you would like to modify the behavior of the parsing behavior beyond the default, you can consult the following webpage for all available options:

<https://dynamorio.org/sec_drcachesim_ops.html>

For example, we can set the number of cores to 9 instead of the default 4 and the cache line size to 19 instead of 64, which will affect the cache simulator, by adding this option when calling DoctorantMemory:

**-additional\_options "-cores 9 -line\_size 19"**

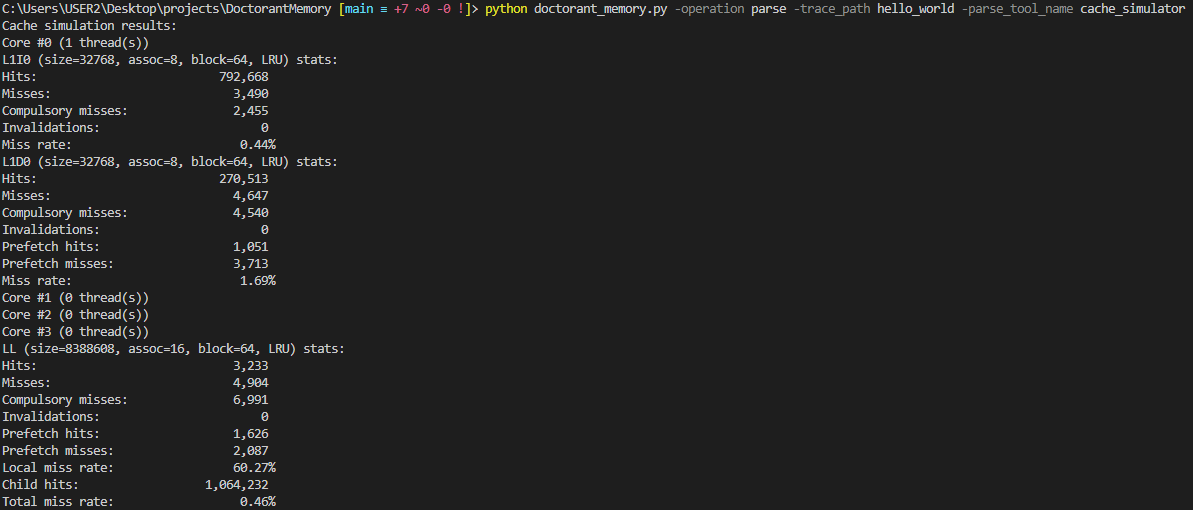
### 3b1. Cache Simulator

The cache simulator tool simulates a multiple leveled cache and shows statistics about cache hits and misses from the memory accesses documented in the trace.

You can use the following command to run it:

**python doctorant\_memory.py -operation parse -trace\_path TRACE\_PATH -parse\_tool\_name cache\_simulator**

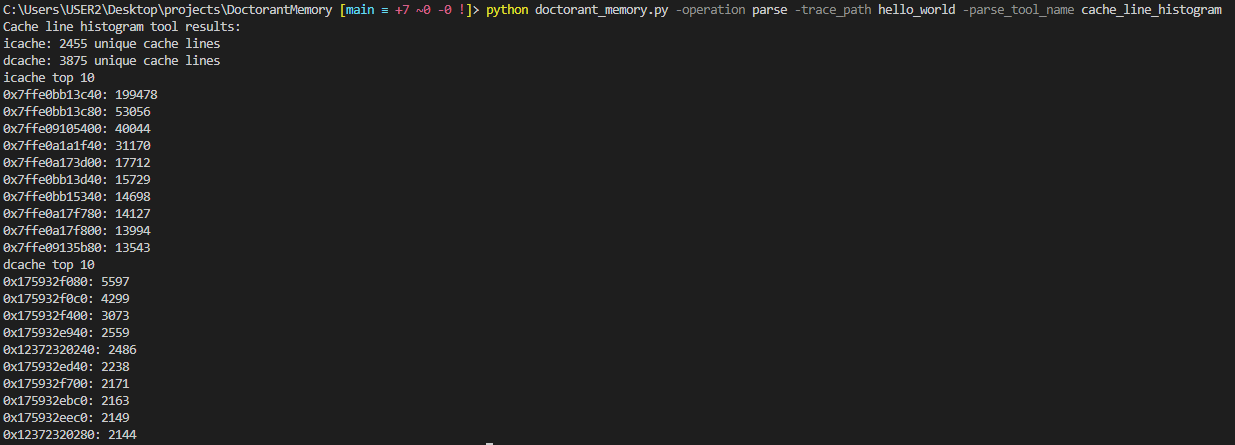
don’t forget to use the name created for your trace in the command. Below is included the output of the tool when using the trace of HelloWorld:



### 3b2. Cache Line Histogram

The cache line histogram tool shows the top 10 addresses in the instruction cache and data cache, as well as the number of unique cache lines.

Below is the output when using the tool:

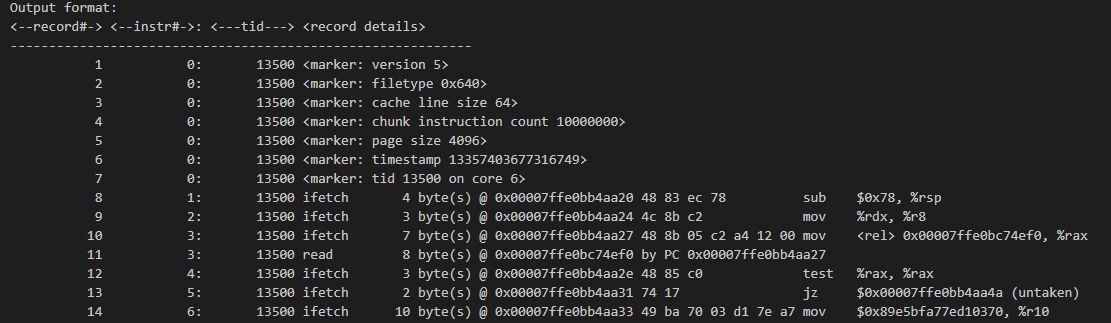
**python doctorant\_memory.py -operation parse -trace\_path TRACE\_PATH -parse\_tool\_name cache\_****line\_histogram**

### 3b3. Human Readable Trace – DrCacheSim’s Format

You can display the human readable trace as generated by DrCacheSim with no additional processing applied.

**python doctorant\_memory.py -operation parse -trace\_path TRACE\_PATH -parse\_tool\_name memory\_accesses\_drcachesim**

The output is included below:



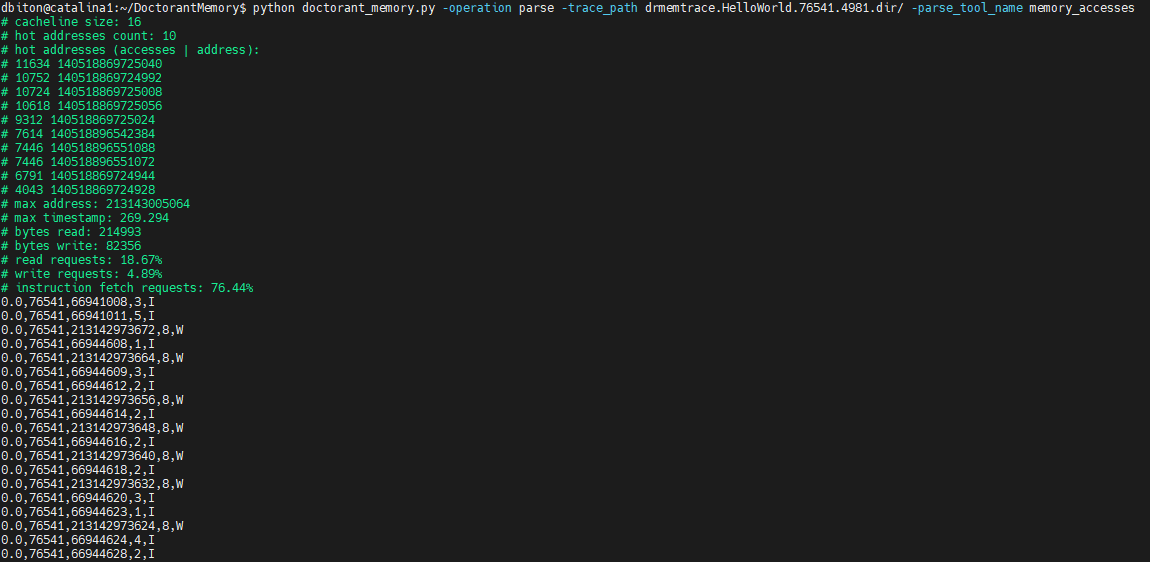
### 3b4. Human Readable Trace – Our Format

The main function of DoctorantMemory, this tool displays each memory access in order, and in addition displays useful statistics in a header (shown below in green). Each memory access is displayed in the following format:

Timestamp, thread id, memory address, request size, access type

The timestamp is nanoseconds since the beginning of execution, the memory address is the number of bytes from the lowest memory address (as a decimal integer), the request size is in bytes, and the access type is W, R, or IF for write, read or instruction fetch.

**python doctorant\_memory.py -operation parse -trace\_path TRACE\_PATH -parse\_tool\_name memory\_accesses**



You can use the **-keep\_logs** flag to keep a file containing the console output, as well as a file containing all of the hot addresses sorted in a descending order, named DOCTORANT\_MEMORY\_HOT\_ADDRESSES\_TS with TS being a timestamp.

You can use the **-parse\_ignore\_inst** flag to ignore instruction fetch memory accesses, as well as using the **-parse\_hot\_addresses\_count** and **-parse\_alignment\_size** (which sets the number of bytes per cache line) like this:

**python doctorant\_memory.py -operation parse -trace\_path TRACE\_PATH**

**-parse\_ignore\_inst -parse\_tool\_name memory\_accesses -parse\_hot\_addresses\_count**

1. **-parse\_alignment\_size 19**

# **Evaluation**

The following runtimes were measured on the catalina1 server.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **cache\_simulator** | **cache\_line\_histogram** | **memory\_accesses** | **memory\_accesses\_drcachesim** |
| **Hello**  **World** | 0.5 Second | 0.5 Second | 12 Seconds | 12 Seconds |
| **Big Trace** | 2:30 Minutes | 3 Minutes | 9 Hours | 7 Hours |
| **Mid Trace** | 4 Seconds | 3 Seconds | 5 Minutes | 2:45 Minutes |

**Hello World:** The trace result of the HelloWorld program we have created. 150KB compressed, 200KB uncompressed.

**Big Trace:** 17571657100049929577.1006531.memtrace.gz from Google’s gsutil Charlie, trace-1. 600MB compressed, 5GB uncompressed.

**Mid Trace:** 17571657100049929577.931429.memtrace.gz from Google’s gsutil Charlie, trace-1. 12MB compressed, 76MB uncompressed.